CS25100: Data Structures and Algorithms, Spring 2017

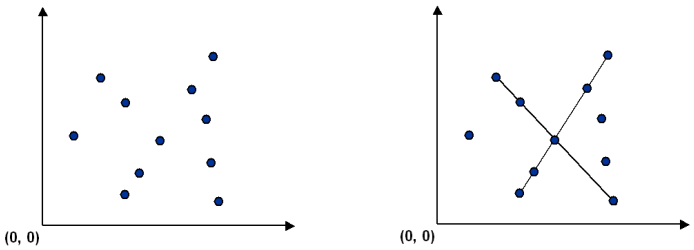
Project 2, Pattern Recognition

Description

Write a program to recognize line patterns in a given set of points.

Computer vision involves analyzing patterns in visual images and reconstructing the real-world objects that produced them. The process is often broken up into two phases: *feature detection* and *pattern recognition*. Feature detection involves selecting important features of the image; pattern recognition involves discovering patterns in the features. We will investigate a particularly clean pattern recognition problem involving points and line segments. This kind of pattern recognition arises in many other applications, for example statistical data analysis.

The problem. Given a set of *N* points in the plane, draw every line segment that connects 4 or more distinct points in the set.



Brute force. Write a program `Brute.java/cpp` that examines 4 points at a time and checks if they all lie on the same line segment, printing out any such line segments to standard output and a file called "visualPoints.txt". To get started, you may use the class Point.java/Point.h (starter code) as the data structure for a point and the client program PointPlotter.java (not graded, simply for visualization) which reads in a list of points from standard input, the output file "visualPoints.txt" and plots them along with the line segments. You will need to supply additional methods in `Point.java/Point.h` which will be used in Brute.java/cpp, including checking whether three or four points lie on the same line. Be sure to implement at least the following API:

JAVA API

```
public class Point implements Comparable{
   public Point(int x, int y)                                          // construct the point (x, y)
   public static boolean areCollinear(Point p, Point q, Point r)       // are the three points collinear?
   public static boolean areCollinear(Point p, Point q, Point r, Point s)  // are the four points collinear?
   public int compareTo(Point that)                                    // is this point lexicographically smaller than that one?
}
```

You will also need to add a user-defined `Comparator` to the `Point` data type to be used for sorting in the second part of the assignment.

C++ API

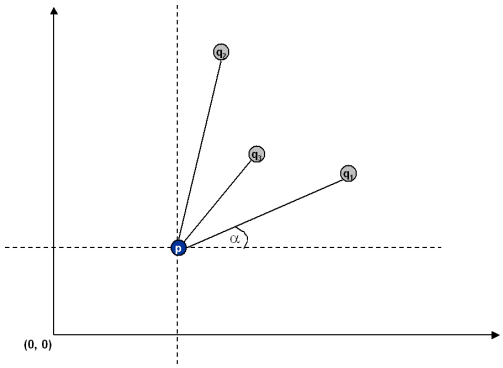
```
struct Point {
    Point(int x, int y);                    // construct the point (x, y)

    bool operator<(const Point& d) const;   // is this point lexicographically
                                            // smaller than that one?

    static bool areCollinear(const Point& p, const Point& q, const Point& r);
                                            // are the three points collinear?
    static bool areCollinear(const Point& p, const Point& q, const Point& r, const Point& s);
                                            // are the four points collinear?

    int x;
    int y;
};
```

You will also need to add a user-defined comparison `functor` for the `Point` data type to be used for sorting in the second part of the assignment.

A sorting solution. Remarkably, it is possible to solve the problem much faster than the brute-force solution described above. Given a point `p`, the following method determines whether `p` participates in a set of 4 or more collinear points.

- Think of `p` as the origin.
- For each other point `q`, determine the angle it makes with `p`.
- Sort the points according to the angle they makes with `p`.
- Check if any 3 (or more) adjacent points in the sorted order have equal angles with `p`. If so, these points, together with `p`, are collinear.

Applying this method for each of the *N* points in turn yields an efficient algorithm to the problem. The algorithm solves the problem because points that make the same angle with `p` are collinear, and sorting brings such points together. The algorithm is fast because the bottleneck operation is sorting.



Write a program `Fast.java` that implements this algorithm using `Arrays.sort()` and a user-defined `Comparator` for `Point` objects. Alternatively, write a program `Fast.cpp` that implements this algorithm using `std::sort() function in C++` and a user-defined `Comparator` for `Point` objects. Your program should use space proportional to *N*. The driver methods should be present only in Brute.java/cpp and Fast.java/cpp

Input format for Fast and Brute versions. The data file consists of an integer *N*, followed by *N* pairs of integers (*x*, *y*), each between 0 and 32,767.

```
% more input6.txt
6
19000  10000
18000  10000
32000  10000
21000  10000
 1234   5678
14000  10000
```

```
% more input8.txt
8
10000      0
    0  10000
 3000   7000
 7000   3000
20000  21000
 3000   4000
14000  15000
 6000   7000
```

Output format. Print to standard output(and to the file "visualPoints.txt") the line segments that your program discovers in the format below (number of collinear points in the line segment, followed by the points **in the order in which they appear on the line segment**. The order meant here is the lexicographical order on the points coordinates that puts points in ascending order first by x coordinate and then followed by y coordinate (if x coordinates are equal)
Please refer to the sample output files in the starter code workspace and stick to the formatting.

```
Java
Windows: % javac -classpath ".;stdlib.jar;" Point.java Brute.java
Unix:    % javac -classpath .:stdlib.jar Brute.java

C++
Unix:    % g++ -std=c++11 Brute.cpp -o Brute

Java
Windows: % java -classpath ".;stdlib.jar;" Brute < input8.txt
Unix:    % java -classpath .:stdlib.jar Brute < input8.txt

C++
Unix:    % ./Brute < input8.txt


4: (0, 10000) -> (3000, 7000) -> (7000, 3000) -> (10000, 0)
4: (3000, 4000) -> (6000, 7000) -> (14000, 15000) -> (20000, 21000)

Java
Windows: % javac -classpath ".;stdlib.jar;" Fast.java
Unix:    % javac -classpath .:stdlib.jar Fast.java

C++
Unix:    % g++ -std=c++11 Fast.cpp -o Fast


Java
Windows: % java -classpath ".;stdlib.jar;" Fast < input6.txt
Unix:    % javac -classpath .:stdlib.jar Fast < input6.txt

C++
Unix:    % ./Fast < input6.txt


5: (14000, 10000) -> (18000, 10000) -> (19000, 10000) -> (21000, 10000) -> (32000, 10000)
```

You may visualize your output using PointPlotter.java. The compilation and execution instructions are in the file itself. After executing either Fast or Brute (in java or C++), you will have an output file called "visualPoints.txt". Now with the file used as input to Fast or Brute program in the last step, use PointPlotter.java to visualize your input and output.

For full credit, do not print *permutations* of points on a line segment (**Your points in the output should be sorted on x axis followed by y axis**). Also, for full credit in `Fast`, do not print *subsegments* of a line segment containing 5 or more points (*e.g.*, if you output $p \to q \to r \to s \to t$, do not also output $p \to q \to s \to t$ or $q \to r \to s \to t$); you have to print out subsegments in `Brute.java` as it's for 4 points.

Report

Submit a report (PDF file) answering the following questions:

- Estimate (using tilde notation) the running time (in seconds) of your two programs as a function of the number of points *N*. Provide a formal analysis about why you consider the formula provided is correct; you can use a style similar to the proof sketch format used in several sections of the book.
- Show, using empirical evidence, the execution time behavior of both versions (brute VS fast). Provide at least a plot and table where you compare the running time observed for different values of *N* (for example, 10, 20, 50, 100, 200, 400, 1000, 2000, 4000, 10000, etc). You are allowed to test different values of N as long as they don't exceed around 200 seconds of running time. The table should contain three columns: *N*, brute-force time and fast time. The plot should have the x-axis be the *N* value and the y-axis will represent running times, where you will show two lines (one for each version).
- Estimate how long it would take to solve an instance of size $N = 1{,}000{,}000$ for each of the two algorithms using your computer.

Submission

Submit your solution before Feb 20 11:59pm. Vocareum will be used to submit assignments.

Directly upload all source code used and libraries (as done in Project 1). The report should be included too. Finally, provide a README file with your name and anything you would like us to know about your program (like errors, special conditions, etc).

Based on an assignment developed by Bob Sedgewick and Kevin Wayne.